Structured Floating-point Exception Handling

An Application Guide

Table of Contents

Introduction

Exception handlers are used to respond to specific errors or exceptions in a program. The exception handling syntax allows the program to filter out all exceptions that are not understood by the program. The ignored exceptions should be passed to other handlers (such as a default handler) that are written to look for those specific exceptions. There are five types of floating-point exceptions that are signaled and can be determined by checking the flags in the Floating-point Status Register (FPSR). The exceptions that are detected by the system and signaled are *Invalid operation, Division by Zero, Overflow, Underflow, Inexact and Denormal*. Each of these exceptions can be associated to a trap under user control by specifying a handler for it. The default response to an exception is to proceed without a trap, unless enabled. If an exception occurs and the associated trap is enabled, the execution of the program in which the exception occurred will be suspended and control transferred to the trap handler previously specified by the user. If no trap handler has been specified, the system's default handler will be activated. This paper discusses floating-point exceptions and the handling of these exceptions using Structured Exception Handling (SEH).

Structured Exception Handling (SEH)

SEH is primarily designed for the C programming language, but works with both C and C++ source files. However, it is not specifically designed for C++ and is not recommended for use in C++ code. Using C++ exception handling guarantees portability in C++ code. Also, the C++ exception handling mechanism is more flexible and can be exercised using the `try-catch-throw` syntax. Having discussed the advantages of C++ exception handling, the primary exception handling mechanism chosen in this document is SEH. As the intent of this document is to address all programming languages, later sections provide tips and techniques to incorporate SEH in FORTRAN applications.

SEH enables a program to separate the main job from the error handling chores. This division allows the program to contain its normal sequence of code and focus on the expected or possible errors later. SEH can be used to make the application more robust. The compiler handles the burden of making SEH work by generating special code when exception blocks are entered and exited from. The compiler produces tables of support data structures to handle SEH and must supply callback functions that the operating system can call. Handling mechanisms are actually of two kinds:

- Exception handlers – which respond to or dismiss the exception and,
- Termination handlers – which are called when an exception causes termination inside a block of code.

These two types of handlers are distinct, yet closely related through a process of unwinding the stack. This section discusses the different ways exception handlers can be written with the help of system provided filters and the structured exception handling syntax used by Microsoft Visual C++* environment. When an exception occurs, Microsoft Windows* 95/98/2000 and Windows NT* look for the most recently installed exception handler that is active. The handler can do one of the following things:

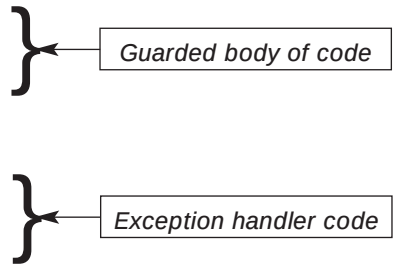
- Pass control to other handlers (ignore the exception in the current handler)
- Recognize but dismiss the exception
- Recognize and handle the exception

Exception handlers usually consist of a filter that recognizes an exception that is to be handled and the actual handler code itself. The filter performs all of the actions described above except handling the exception itself. The next section describes the role of filters and writing exception handlers using the SEH syntax supported in Microsoft Visual C++.

Exception handlers and filters

Exception handlers allow applications to process both hardware and software exceptions. The *try-except* pair in Microsoft Visual C++ achieves this. Consider the code shown below:

```
__try  {
            Code block 1
}
__except ( filter-expression  )  {
            Code block 2
}
```



This code shows an example of an exception handler using the **try-except** block. The **try** key word tells the compiler to monitor any of the code in the try block for exceptions. The block following try is called the *guarded body of code.* Immediately following except is a set of parentheses that contain the *filter expression,* which is generally a constant value or a function call that returns a constant value. Following the filter expression is a block of code that is executed by the system depending on the outcome of the filter and is called the *exception handler.* The code contained in the try block should be sequential and not contain any **break**, **continue**, **goto** or **return** statements because these are treated and handled like exceptions.

When an exception occurs in *code block 1,* the exception handler takes and evaluates the filter. The primary function of a filter is to analyze the exception situation and determine the appropriate value to return. The system filter (**_fpieee_flt**) under Microsoft Windows 95/98/2000 and Windows NT provides the easiest mechanism to handle the floating-point exceptions. On an exception, the **_fpieee_flt** filter extracts the values of the operands that caused the exception to occur and using a software emulator, computes the value of the resultant. This feature of setting the resultant value in the **_FPIEEE_RECORD** data structure is true only on Microsoft Windows NT systems. Newer processor packs of Microsoft Visual C++ contain runtime libraries that include this feature on Microsoft Windows 98/NT systems. Using this filter allows the application to handle all types of floating-point exceptions caused, including those caused by using the Streaming SIMD Extension instructions available on Intel® Pentium® III and the SIMD instructions available on Intel® Itanium™ systems.

The filter extracts the relevant information from the available exception data structures and prepares a new data structure, **_FPIEEE_RECORD** that can be made available by making a local copy to any exception handler written by the user. If a user handler is not provided, control is transferred to a default handler.

The next action depends on the value returned by the filter which can take any of the following 3 possible values:

- **EXCEPTION_CONTINUE_SEARCH (0)** – Passes control to exception handler with next highest precedence. The handler has declined to recognize the exception and the system tries to find another exception handler on the stack.
- **EXCEPTION_CONTINUE_EXECUTION (-1)** – Dismisses exception, and continues execution at the location where the exception was raised.
- **EXCEPTION_EXECUTE_HANDLER (1)** – Handles exception by executing statements in *Code block 2.*

If the value returned by filter is EXCEPTION_EXECUTE_HANDLER, execution does not resume where the exception was raised, but falls through to *Code block 2.* All blocks and function calls nested in *Code block 1* are terminated. The *Code block 2* of the exception handler can be used to print out any meaningful debug information about the kind of exception that occurred and where it occurred. However, calls to getting exception information from the system can only be made when evaluating the filter expression. In order for this information to be available in the handler, the exception data structures will have to be saved to local variables that can be used at a later time.

Using the system exception filter

The system exception filter for floating-point exceptions requires a user defined trap handler to handle these exceptions. This section describes the syntax and a simple example that uses the exception filter along with a trap handler. Consider the code shown below. Bold face text shows the floating-point exception that is being recognized. In this case, the underflow exception is recovered from by setting the result to an acceptable value. All other floating-point exceptions are handled in this example.

```
#include <fpieee.h>
#include <excpt.h>
#include <float.h>

int user_trap_handler( _FPIEEE_RECORD * );

int user_trap_handler( _FPIEEE_RECORD *pieee )
{
   // user-defined ieee trap handler routine:
   // there is one handler for all
   // IEEE exceptions
   // Assumes the user wants all underflow
   // exceptions in single precision to return 0.

      if( pieee->Cause.Overflow ) {
        printf("Exception Cause: Overflow\n");
      }
      else if( pieee->Cause.Underflow ) {
        printf("Exception Cause: Underflow\n");
```

```
            switch( pieee->Result.Format) {
            //
            // Returning EXCEPTION_CONTINUE_EXCEPTION causes this exception
            // to be dismissed after the excepting result value is changed
            // to an acceptable value, hence recoverable
            //
            case _FpFormatFp32: pieee->Result.Value.Fp32Value = 0.0F;
                                return EXCEPTION_CONTINUE_EXECUTION;
                                break;
            case _FpFormatFp64: break;
            case _FpFormatFp80: break;
            case _FpFormatFp82: break;
            case _FpFormatFp128: break;
            default: printf ("ERROR: Result.Format Unknown = %d\n",
                              pieee->Result.Format);
                //
                // Returning EXCEPTION_EXECUTE_HANDLER causes this exception
                // to be handled in code block 2 as no recovery code is
                // present to recover from the exception and continue
                // execution
                //
                return EXCEPTION_EXECUTE_HANDLER;
                break;
              }
          }
          else if( pieee->Cause.Inexact ) {
            printf("Exception Cause: Inexact\n");
          }
          else if( pieee->Cause.ZeroDivide ) {
              printf("Exception Cause: ZeroDivide\n");

          }
          else if( pieee->Cause.InvalidOperation ) {
              printf("Exception Cause: InvalidOperation\n");
          }
        return EXCEPTION_EXECUTE_HANDLER;
}

#define _EXC_MASK    \
    _EM_INVALID    + \
    _EM_OVERFLOW   + \
    _EM_ZERODIVIDE + \
    _EM_INEXACT

void main( void )
{
   . . . . .
   // ...
   __try {
      // unmask all fp exceptions except UNDERFLOW here
      // Code that may generate FP exceptions appears after these comments
   }
   __except ( _fpieee_flt( GetExceptionCode(),
                           GetExceptionInformation(),
                           user_trap_handler ) ){
      // code that gets control
      // If fpieee_flt handler returns
      // EXCEPTION_EXECUTE_HANDLER then control is
      // transferred to this section.
```

Code Block 2

```
        // Include architecture specific output using
         // #ifdef's for the architectures supported.
      }
      // ...
      . . . . .
}
```

In the above example, if the *underflow* exception occurs, the exception is recovered from by setting the result to a value (zero, in the example) that is acceptable by the application and continues from the place the exception occurred. If other floating-point exceptions are unmasked in the above example, the handler will print the type of exception encountered, handle the exception and execution falls through to *Code block 2*. This mechanism is the easiest to implement as the filter function is made available by the system.

The **_FPIEEE_RECORD** that is the argument to the trap handler contains information about the operands that caused the exception and the result operand and their respective formats. This provides adequate debug information in most cases. However, some applications may require additional debug information to track the exception that has occurred. This can be achieved by modifying the exception filter function as described in the next section.

Modifying the exception filter for more debug information

To access additional debug information (Exception address, Excepting instruction, stack pointer, integer and floating-point register information, etc.) the exception filter will have to be modified to retain the information that is returned by `GetExceptionInformation()` in local variables for later access. This allows all debug information that is desired to be available when handled. With these modifications (shown in bold face font) the except block of the code shown above looks like:

```
//
// These local variables within the scope of try-except
// pair are used to retain the exception information
// data structures for later use
//
EXCEPTION_RECORD    ExceptionRecord;
CONTEXT             ExceptionContext;



__except ( exception_flt_debug( GetExceptionCode(),
                                GetExceptionInformation(),
                                user_trap_handler,
                                &ExceptionRecord,
                                &ExceptionContext) ){

      // code that gets control if fpieee_handler returns
      // EXCEPTION_EXECUTE_HANDLER goes here
      // Use ExceptionRecord and ExceptionContext here to print
      // additional debug information
      //
      // e.g.:- ExceptionAddress =
      //             (char *)ExceptionRecord.ExceptionAddress;
      //
      // See Appendix A for details on the function used below
```

```
      //

      printExceptionDebugInformation(&ExceptionRecord, &ExceptionContext);
}

//
//  The modified filter function accepts two new arguments
//  that are used to retain the exception information.
//
exception_flt_debug(unsigned long eXceptionCode,
                    PEXCEPTION_POINTERS p,
                    int (*handler)(_FPIEEE_RECORD *),
                    EXCEPTION_RECORD *ExceptionRecord,   // local var1
                    CONTEXT          *ExceptionContext ) // local var2
{

   //
   // Function start.  Copy the exception information
   // to local areas for later processing.
   //
   (void) memcpy ( (void *) ExceptionRecord,
                   (void *) p->ExceptionRecord,
                   sizeof(EXCEPTION_RECORD) );
   (void) memcpy ( (void *) ExceptionContext,
                   (void *) p->ContextRecord,
                   sizeof(CONTEXT) );

   //
   // Make a call to the system  exception filter
   // for handling all the exceptions
   //
   return _fpieee_flt(eXceptionCode,  p, handler) ;
}
```

The drawback of this approach is that if the application code generates exceptions other than floating-point exceptions, new handlers will have to be written and try-except blocks nested or code separated into different try-except blocks. This is avoided by writing a single exception handler that recognizes all the important exceptions that are generated by the application and handles them appropriately.

Handling all exceptions using the same handler

In cases where there is a need for exceptions other than floating-point to be handled, the above mechanism will not be sufficient and a filter has be written by the application developer customized for their application. A mechanism to furnish such a filter with reduced coding effort can be achieved by calling the system filter for all floating-point exceptions, but one that contains custom filtering for all other exceptions. Using this approach, as shown in the example below, the filter can be used to handle user defined and other system exceptions in addition to the floating-point exceptions that could occur in an application.

```
main ()
{
    EXCEPTION_RECORD  ExceptionRecord
    CONTEXT           ExceptionContext;
    __try {
...
}
__except ( Eval_Exception( GetExceptionCode(),
                           GetExceptionInformation(),
                           user_trap_handler,
                           &ExceptionRecord,
                           &ExceptionContext) ) {
      // code that gets control
      // if filter returns EXCEPTION_EXECUTE_HANDLER control
      // is transferred to this block
      // Use ExceptionRecord and ExceptionContext  here to print
      // additional debug information
      //
      // eg:- ExceptionAddress =
      //     (char *)ExceptionRecord.ExceptionAddress;

      printExceptionDebugInformation(&ExceptionRecord, &ExceptionContext);
    }
    ...
}

int Eval_Exception (int eXceptionCode, PEXCEPTION_POINTERS p,
                    int (*handler)(_FPIEEE_RECORD *),
                    EXCEPTION_RECORD *ExceptionRecord,
                    CONTEXT          *ExceptionContext )
{
   /* Function start.  Copy the exception information
    * to local areas for later processing.
    */
   (void) memcpy ( (void *) ExceptionRecord,
                   (void *) p->ExceptionRecord,
                   sizeof(EXCEPTION_RECORD) );
   (void) memcpy ( (void *) ExceptionContext,
                   (void *) p->ContextRecord,
                   sizeof(CONTEXT) );

    switch( p->ExceptionRecord->ExceptionCode )  {

        case EXCEPTION_ACCESS_VIOLATION:
           // Ignore this exception and pass it
           // to the default handler
           return EXCEPTION_CONTINUE_SEARCH;
           break;
        case MY_EXCEPTION_1[1]:
           // Add clean up code here !!
           // continue execution from the instruction after
           // the instruction that caused the exception
           return EXCEPTION_CONTINUE_EXECUTION;
           break;
        case MY_EXCEPTION_2[1]:
           // This exception may be a severe exception.
           // Execute the handler code and abort normal
           // program flow
```

```
            return EXCEPTION_EXECUTE_HANDLER;
            break;
         case EXCEPTION_FLT_OVERFLOW:
         case EXCEPTION_FLT_UNDERFLOW:
         case EXCEPTION_FLT_DENORMAL_OPERAND:
         case EXCEPTION_FLT_DIVIDE_BY_ZERO:
         case EXCEPTION_FLT_INEXACT_RESULT:
         case EXCEPTION_FLT_INVALID_OPERATION:

            // filter all floating-point exceptions through
            // the system provided filter

            return _fpieee_flt(eXceptionCode, p, handler);

            break;
         default:
            // Ignore unknown exceptions and pass it
            // to the default handler
            return EXCEPTION_CONTINUE_SEARCH;
            break;
      }

}
```

[1]MY_EXCEPTION_1 and MY_EXCEPTION_2 can be generated by using the **RaiseException()** call.

It is a good idea to use a function call in the *filter* expression whenever *filter* does anything complex. Evaluating the expression causes execution of the function, in this case, **Eval_Exception.** Using the framework provided above, write one custom filter to handle all exceptions generated by the application. Also, having a single exception handler and filter function makes it easier to maintain and support new architectures. Appendix A discusses an exception handler that prints out IA-64 specific debug information.

Tips for handling exceptions in FORTRAN code using SEH

Since FORTRAN 77 or FORTRAN 90 do not support structured exception handling, use a C/C++ driver with the Fortran code. The driver has a main function that consists of a try-except block and calls a Fortran driver function that encompasses calls to all sections of the code that have to be observed for exceptions. Using this approach, a Fortran application can benefit from the SEH that is available under Microsoft Visual C/C++ environment.

FORTRAN Code

```
PROGRAM MYTEST (. . . .)
. . . . .
. . . . .
. . . . .
STOP
END
```

Modified FORTRAN Code

```
SUBROUTINE MYTEST (. . . .)
. . . . .
RETURN
END

main (int argc, char ** argy)
{
     . . . .
     __try {
           MYTEST (. . .)
       }
     __except (. . . . .)   {
       }
     . . . .
}
```

PROGRAM **MYTEST** changed to a SUBROUTINE

C Driver

Summary

Through the use of Structured Exception Handling (SEH) and a single exception handler to handle all exceptions generated by an application, developers can write code that is easy to port among various architectures that support Microsoft Windows* OS families and is relatively easy to maintain. The error handling code is isolated to one module that is easy to modify to include new machine types and the filter function is easy to expand to support new exceptions. This helps avoid repeating the filter and handling code throughout the application.

Appendix: IA-64 Debug Information

This section shows the code that is used for printing out **IA-64 specific** debug information while handling floating-point exceptions. The function `printExceptionDebugInformation()` can be called in `__except` block of the code (or Code *block* 2) to print the relevant debug information when exceptions are handled. To extract information for other architecture types, similar code can be written by examining the CONTEXT data structure for the corresponding architecture and enclosing the code in #ifdef - #endif pairs.

```
void  printExceptionDebugInformation( EXCEPTION_RECORD  *ExceptionRecord
                                      CONTEXT           *Context )
    {
          __int64 reg, iRegValue;
          _FP128  regValue;
          int     i;

          char *StackAddress;

          reg = GetIntegerRegister(12 /*Stack Pointer is in Register 12*/);
          StackAddress = (char *)reg;

          printf ("StackAddress         : %8x %8x\n",
                  (int)((__int64)StackAddress >> 32) & 0xffffffff,
                  (int)StackAddress & 0xffffffff);

          //
          //  Print the floating-point status register showing the
          //  traps enabled and the exception that have occurred
          //
          FPSR = Context->StFPSR;
          printf ("FPSR                 : %8x %8x\n",
             (int)(FPSR >> 32) & 0xffffffff, (int)FPSR & 0xffffffff);
          printf ("FPSR  (Traps enabled): %8x\n", (int)FPSR & 0x0000001f);
          printf ("FPSR  (Exceptions)   : %8x\n", (int)FPSR & 0x000ff000);

          ExceptionAddress = (char *)ExceptionRecord->ExceptionAddress;
          printf ("ExceptionAddress     : %8x %8x\n",
             (int)((__int64)ExceptionAddress >> 32) & 0xffffffff,
             (int)ExceptionAddress & 0xffffffff);

          if (ExceptionRecord->ExceptionInformation[0]) {
           printf ("ERROR: software generated exception\n");
          }

          //
          // Print all the integer registers
          //
          for(i = 0; i <128; ++i) {
           iRegValue = GetIntegerRegister(Context, i);
           printf("Integer Reg %.2d –> 0x%p\n", i, iRegValue);
          }
```

```
      //
      // Print all the floating-point registers
      //
      for(i = 0; i <128; ++i) {
       regValue = GetFloatingPointRegister(Context, i);
       printf("FP Reg %.3d –> %8.8x %8.8x %8.8x %8.8x\n", i,
          regValue.W[0], regValue.W[1], regValue.W[2], regValue.W[3]);
      }
}

//
// This function is called in GetExceptionDebugInformation() to print
// the values of all floating-point registers on an IA-64 platform
//

_FP128 GetFloatingPointRegister(CONTEXT *Context, unsigned int fr)
{
    _FP128            RetValue;
    unsigned __int64 *p1, *p2;

    p1 = (unsigned __int64 *)&RetValue;

    if (fr == 0 ) {
      *p1 = 0;                        //  Register 0 is always ZERO
      *(p1 + 1) = 0;
     }
    else if (fr == 1) { /* 1.0 */
      *p1       = 0x8000000000000000;  //  Register 1 is always ONE
      *(p1 + 1) = 0x000000000000ffff;
     }
    else if (fr >= 2 && fr <= 127) {
      p2     = (unsigned __int64 *)&(Context->FltS0);  // First FP register
      p2        =  p2 + 2 * (fr - 2);
      *p1       = *p2;
      *(p1 + 1) = *(p2 + 1);
     }
     else {
      fprintf (stderr, "IEEE Filter/GetFloatingPointRegister () Internal Error:");
      fprintf (stderr, "FP register number f = %x is not valid\n", fr);
      exit (1);
    }

    return (RetValue);
}
```

```
    //
    // This function is called in GetExceptionDebugInformation() to print
    // the values of all integer registers on and IA-64 platform
    //

    __int64 GetIntegerRegister(CONTEXT *Context, unsigned int ir)
    {
         __int64           RetValue;
         unsigned __int64 *p1, *p2;

         p1 = (unsigned __int64 *)&RetValue;

         if (ir == 0 )
          RetValue = 0;                 // Register 0 is always ZERO
         else if (ir >= 1 && ir <= 31) {
             p2  = (unsigned __int64 *)&(Context->IntGp);  // First FP register
          p2  =  p2 + ir - 1;
             *p1 = *p2;
         }
         else {
          fprintf (stderr, “IEEE Filter / GetIntegerRegister () Internal
Error:”);
             fprintf (stderr, “Integer register number i = %x is not valid\n”,
ir);
          exit (1);
         }

         return (RetValue);
    }
```